JIOS, VOL. 40, NO. 2 (2016) SUBMITTED 02/16; ACCEPTED 12/16

UDC 004.423.42
Original scientific Paper

Categorical Model of Structural Operational Semantics for Imperative Language

William Steingartner *william.steingartner@tuke.sk*
Faculty of Electrical Engineering and Informatics
Technical university of Košice
Letná 9, 042 00 Košice, Slovak Republic

Valerie Novitzká *valerie.novitzka@tuke.sk*
Faculty of Electrical Engineering and Informatics
Technical university of Košice
Letná 9, 042 00 Košice, Slovak Republic

Abstract

Definition of programming languages consists of the formal definition of syntax and semantics. One of the most popular semantic methods used in various stages of software engineering is structural operational semantics. It describes program behavior in the form of state changes after execution of elementary steps of program. This feature makes structural operational semantics useful for implementation of programming languages and also for verification purposes. In our paper we present a new approach to structural operational semantics. We model behavior of programs in category of states, where objects are states, an abstraction of computer memory and morphisms model state changes, execution of a program in elementary steps. The advantage of using categorical model is its exact mathematical structure with many useful proved properties and its graphical illustration of program behavior as a path, i.e. a composition of morphisms. Our approach is able to accentuate dynamics of structural operational semantics. For simplicity, we assume that data are intuitively typed. Visualization and facility of our model is not only a new model of structural operational semantics of imperative programming languages but it can also serve for education purposes.

Keywords: category theory, morphism, semantic function, state, structural operational semantics

1. Introduction

An important part of the formal definition of a programming language is a definition of its semantics. In this paper we deal with one of the known method, structural operational semantics that provides a simple and direct method for describing meaning of programs written in some programming language. Its advantage is that it requires minimal knowledge of mathematics and it is easily understandable by practical programmers [1], [2], [3]. We present a new approach how to define a model of simple imperative language in the category of states.

There are several semantic methods used simultaneously with structural operational semantics. They differ in used mathematical equipment and the area of usage. One of the first formulated semantic methods is denotational semantics formulated by Scott Strachey in [4] and later by David Schmidt [5]. It requires quite deep knowledge of mathematics because the meaning of programs is expressed by functions from syntactical domains to semantic domains which can be non-trivial mathematical structures, e.g. lattices. Categorical models for denotational semantics is based on category of types [6], [7]. Therefore we cannot be surprised that structural operational semantics gained much more attention in community of programmers than denotational semantics [8].

A simpler semantic method similar to structural operational semantics is natural semantics formulated by Gilles Kahn [9] and is often called semantics of big steps. The author followed two aims:

- to simplify semantic description for software engineers instead of difficult mathematical notations of currying and continuation functions in denotational semantics; and
- to abstract from elementary steps of execution in structural operational semantics.

Natural semantics describes a change of states caused by execution the whole statements [10] and can be useful for specification languages or in program verification [11].

There are known several other semantic methods less or more used in various areas of programming. Axiomatic semantics [12] is based on satisfying postconditions after executing of statements from truth preconditions before this action. The triples precondition, statement and postcondition are called Hoare's formulae. Algebraic semantics [13], [14] specifies abstract data types possibly with parameters and it models them by heterogeneous algebras. Game semantics [15], [16] describes the meaning of programs in the form of game trees and game arenas.

As the author of structural operational semantics is regarded Gordon Plotkin. In his work [17] he formulated this semantic method as a formal tool for describing detailed execution of programs by transition relations between configurations before and after performing elementary steps of operations. The main ideas about his approach and his motivation is explained in [18].

Structural operational semantics generates a labeled transition system for a program written in programming language [19]. This transition system consists of transition

rules describing modification of states. A state is a basic notion of structural operational semantics and it can be considered as some abstraction of computer memory. Every transition rule has its premise or premises and one conclusion. Premises and a conclusion are transitions. The rules can be decorated by additional conditions in premisses in the form of predicates. A rules has a form

$$\frac{\text{premise}_1, \ldots, \text{premise}_n, \text{condition}(s)}{\text{conclusion}}$$

If all premises and all conditions (if exist) are valid before execution of a statement in the conclusion, then a conclusion is valid [20].

As it has been published in [19], structural operational semantics is essentially a description of program behavior. Because it provides a detailed description of program performing, its main application area is in implementation of programming [21]. By the years, this semantic method became very popular among software engineers because of its simplicity. To avoid some restrictions of this method many extensions have been worked out.

Structural operational semantics expresses context dependencies using a new notion of environment. Context dependencies describe the relationships required between the declarations of variables and their usage in nested blocks with respect to scope rules.

In the last decades many new results were published about structural operational semantics. Turi [22] in his PhD. thesis formulated coalgebraic categorical model of this method and he showed its duality with denotational approach. New approaches to operational semantics were published in [23], [24]. Among new research results in the area of this semantic methods belongs also formulation of modular structural operational semantics published in [25], [26], [27].

2. The language $\mathscr{J}ane$

Categorical model of structural operational semantics we define for a sample imperative language $\mathscr{J}ane$. It consists of traditional syntactic constructions of imperative languages, namely arithmetic and Boolean expressions, variable declarations and statements. For defining formal syntax of $\mathscr{J}ane$ we introduce the following syntactic domains:

$n \in \mathbf{Num}$	- digit strings;
$x \in \mathbf{Var}$	- variable names;
$e \in \mathbf{Expr}$	- arithmetic expressions;
$b \in \mathbf{Bexpr}$	- Boolean expressions;
$S \in \mathbf{Statm}$	- statements;
$D \in \mathbf{Decl}$	- sequences of variable declarations.

The elements $n \in \mathbf{Num}$ have no internal structure from semantic point of view.

Similarly, $x \in \mathbf{Var}$ are only variable names without internal structure significant for defining semantics.

The syntactic domain $\mathbf{Expr}$ consists of all well-formed arithmetic expressions constructed by the following production rule:

$$e ::= n \mid x \mid e + e \mid e - e \mid e * e.$$

A Boolean expression from $\mathbf{Bexpr}$ can be of the following structure:

$$b ::= \texttt{false} \mid \texttt{true} \mid e = e \mid e \leq e \mid \neg b \mid b \wedge b.$$

The variables used in programs have to be declared. We consider $D \in \mathbf{Decl}$ as a sequence of declarations:

$$D ::= \texttt{var}\ x; D \mid \varepsilon$$

where ε is the empty sequence. We assume that variables are implicitly of type integer. This restriction enables us to focus on main ideas of our approach.

We consider five Dijkstra's statements as statements in language $S \in \mathbf{Statm}$: assignment, empty statement, sequence of statements, conditional statement and cycle statement together with block statement and input statement:

$$\begin{aligned} S ::= & \\ & x := e \mid \texttt{skip} \mid S; S \mid \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \mid \\ & \texttt{while}\ b\ \texttt{do}\ S \mid \texttt{begin}\ D; S\ \texttt{end} \mid\ \texttt{input}\ x. \end{aligned}$$

3. Specification of states

A state is a basic concept of structural operational semantics. It can be considered as some abstraction of computer memory. Every variable occurring in a program has to be allocated, i.e. some memory cell is reserved for a declared variable and a name of it is assigned to the allocated memory cell. We can assign and modify a value of allocated variable inducing change of state. Because of block structure of Jane, we have to consider also a level of block nesting.

According to previous ideas we formulate the signature Σ_{State} for states. We define abstract data type $State$ using types Var and $Value$ of variables and values. A signature Σ_{State} consists of types and operation specifications on the type $State$:

$$\begin{aligned} \Sigma_{State} = & \\ & \underline{types:} \quad State, Var, Value \\ & \underline{opns:} \quad init :\rightarrow State \\ & \qquad\quad alloc : var, State \rightarrow State \\ & \qquad\quad get : Var, State \rightarrow Value \\ & \qquad\quad del : State \rightarrow State \end{aligned}$$

The operation specifications have the following intuitive meaning:

- $init$ creates a new state, the initial state of a program;
- $alloc$ reserves a new memory cell for a variable in a given state (and nesting level);
- get returns a variable value in an actual state;
- del deallocates (releases) all variables together with their values on a given nesting level.

The signature Σ_{State} serves as a basis for constructing our model of $\mathscr{J}ane$ as the category of states.

4. Representation of states

We construct operational model of $\mathscr{J}ane$ as the category $\mathscr{C}_{State}$ of states. First, we assign to states their representation. We assign to the type name $Value$ the set of integer numbers set $\mathbb{Z}$:

$$\mathbf{Value} = \mathbb{Z}.$$

For undefined values we use the symbol $\perp$. The type name Var is represented by the set $\mathbf{Var}$ of variable names. Our representation of an element of type $State$ has to express a variable name and its value with respect to actual nesting level. Let $\mathbf{Level}$ be a finite set of nesting levels denoted by natural numbers l:

$$l \in \mathbf{Level}, \;\; \mathbf{Level} = \mathbb{N}.$$

We assign to the type $State$ the set $\mathbf{State}$ of states. Now, we can represent every state $s \in \mathbf{State}$ as a function

$$s : \mathbf{Var} \times \mathbf{Level} \rightharpoonup \mathbf{Value}. \tag{1}$$

This function is partially defined, because a declaration does not assign a value to declared variable. Every state s expresses one step of program execution. Our definition of states can be also considered as a table with possibly empty cells denoted by $\perp$.

Every state s can be expressed as a sequence:

$$s = \langle ((x, 1), v_1), \ldots, ((z, l), v_n) \rangle$$

of ordered triples

$$((x, l), v),$$

where (x, l) is declared variable x on a nesting level l with actual (possibly undefined) value v. Another representation of state is a table which contains names of variables, the level of their declarations and actual values stored in the variables (Figure 1).

variable	level	value
x	1	v_1
⋮		
z	l	v_n

Figure 1: Representation of a state by table

The first declared variable x in this table has nesting level 1 and value v_1. Remaining declared variables have corresponding identifications in this state.

Now, we can define the representation of operations from Σ_{State} as follows. The operation $[\![init]\!]$ defined by

$$[\![init]\!] = s_0 = \langle((\bot, 1), \bot)\rangle$$

creates the initial state s_0 of a program with no declared variable. Its role is to set nesting level to value 1 (Figure 2).

variable	level	value
$\bot$	1	$\bot$

Figure 2: Initial state of the program

The operation $[\![alloc]\!]$ is defined by

$$[\![alloc]\!](x, s) = s \diamond \langle((x, l), \bot)\rangle,$$

where '$\diamond$' is concatenation of sequences. This operation sets actual nesting level to declared variable. Because of undefined value of a variable within declaration, the operation $[\![alloc]\!]$ does not change the state (Figure 3).

variable	level	value
⋮	⋮	⋮
x	l	$\bot$

Figure 3: Allocation of a new variable

The operation $[\![get]\!]$ returns a value of a variable declared on the highest nesting level and can be defined by

$$[\![get]\!](x, \langle \ldots, ((x, l_i), v_j), \ldots, ((x, l_k), v_{k'}), \ldots \rangle) = v_{k'},$$

where $l_i < l_k$.
The operation $[\![del]\!]$ deallocates (forgets) all variables declared on the highest nesting level l_j (Figure 4):

$$[\![del]\!](s \diamond \langle ((x_i, l_j), v_k), \ldots, ((x_n, l_j), v_m) \rangle) = s.$$

variable	level	value
⋮	⋮	⋮
x	l_{j-1}	v
x_i	l_j	v_k
⋮	⋮	⋮
x_n	l_j	v_m

Figure 4: Deallocation of all variables declared on the level l_j

We construct the category $\mathscr{C}_{State}$ as a category of states. Category objects are states s defined above with special object $s_\perp = \langle ((\perp, \perp), \perp) \rangle$ expressing an undefined state.

Category morphisms express change of states caused by execution of statements and they will be defined later.

5. Arithmetic and Boolean expressions

Arithmetic and Boolean expressions serve for computing values of two implicit types of the language $\mathscr{J}ane$. In defining semantics of both types of expressions, an actual state is used but not changed in the process of evaluation. The following tables (Table 1 and Table 2) define semantic functions together with corresponding semantic operations for arithmetic and Boolean expressions over the semantic domain of states.

$$[\![e]\!] : \mathbf{State} \to \mathbf{Value}.$$

We note that $\oplus$, $\ominus$ and $\otimes$ occurring on the right of these equations are the usual arithmetic operations, whilst on the left they are just pieces of syntax.

$$[\![b]\!] : \mathbf{State} \to \mathbf{Bool}.$$

$$[\![n]\!]s = \mathbf{n}$$

$$[\![x]\!]s = [\![get]\!](x, s)$$

$$[\![e_1 + e_2]\!]s = [\![e_1]\!]s \oplus [\![e_2]\!]s$$

$$[\![e_1 - e_2]\!]s = [\![e_1]\!]s \ominus [\![e_2]\!]s$$

$$[\![e_1 * e_2]\!]s = [\![e_1]\!]s \otimes [\![e_2]\!]s$$

Table 1: Semantics of arithmetic expressions

$$[\![\texttt{true}]\!]s = \mathbf{true}$$

$$[\![\texttt{false}]\!]s = \mathbf{false}$$

$$[\![e_1 = e_2]\!]s = \begin{cases} \mathbf{true} & \text{if } [\![e_1]\!]s{=}[\![e_2]\!]s \\ \mathbf{false} & \text{otherwise} \end{cases}$$

$$[\![e_1 \leq e_2]\!]s = \begin{cases} \mathbf{true} & \text{if } [\![e_1]\!]s{\leq}[\![e_2]\!]s \\ \mathbf{false} & \text{otherwise} \end{cases}$$

$$[\![\neg b]\!]s = \begin{cases} \mathbf{true} & \text{if } [\![\neg b]\!]s{=}\mathbf{false} \\ \mathbf{false} & \text{otherwise} \end{cases}$$

$$[\![b_1 \wedge b_2]\!]s = \begin{cases} \mathbf{true} & \text{if } [\![b_1]\!]s{=}[\![b_2]\!]s{=}\mathbf{true} \\ \mathbf{false} & \text{otherwise} \end{cases}$$

Table 2: Semantics of Boolean expressions

Evaluation of expressions has no effect on states, i.e. objects of our category $\mathscr{C}_{State}$. **Value** and **Bool** are semantic domains for integers and Booleans, resp.

$$\mathbf{Value} = \mathbb{Z}, \quad \mathbf{Bool} = \mathbb{B},$$

where $\mathbb{B}$ is the set containing Boolean values, i.e. $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$.

6. Declarations

Every variable occurring in a $\mathscr{J}ane$ program has to be declared. Declarations are elaborated, i.e. a memory cell is allocated and named by a declared variable. Therefore elaboration of a declaration

$$\texttt{var}\ x$$

is represented as an endomorphism:

$$[\![\ ]\!]_D : s \to s$$

for a given state s and defined by

$$[\![\texttt{var}\ x]\!]s = [\![alloc]\!](x, s).$$

A sequence of declarations is represented as a composition of corresponding endomorphisms:

$$[\![\texttt{var}\ x; D]\!]s = [\![D]\!] \circ [\![alloc]\!](x, s).$$

If we consider a state as a table, a declaration creates new (raw) entry for a declared variable with the actual level of nesting and undefined value

$$((x, l), \bot).$$

7. Statements

Statements are the most important constructions of imperative languages. They execute program actions, i.e. they take values from the actual state and provide new values. A state is changed only if a value of allocated variable is modified. This change of state we model in category $\mathscr{C}_{State}$ by morphisms between objects. Let S be a statement. Its semantics is a morphism in category $\mathscr{C}_{State}$:

$$[\![S]\!] : \mathbf{State} \to \mathbf{State}, \tag{2}$$

where s and s' are objects in category $\mathscr{C}_{State}$. Statements are executed in sequence, as they are written in program text. In this contribution we do not consider the statements breaking sequential execution, e.g. `goto` statement or exceptions.

Assignment statement $x := e$ stores a value of arithmetic expression e in a state s in a memory cell allocated for the variable x on maximal (highest) level of nesting. This condition ensures that local variable visible in given scope is used.
The semantics is as follows

$$[\![x := e]\!]s = \begin{cases} s\left[((x, l), v) \mapsto ((x, l), [\![e]\!]s)\right], & \text{for } ((x, l), v) \in s \\ \bot, & \text{otherwise} \end{cases}$$

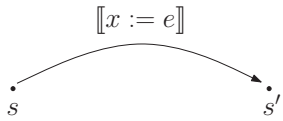

Figure 5: Morphism for assignment

and it is expressed by a morphism in the Fig. 5.
The notation

$$s' = s\,[((x, l), v) \mapsto ((x, l), [\![e]\!]s)] \tag{3}$$

describes a new state s' that is an actualization of the state s in its entry for the variable x which value is changed to $[\![e]\!]s$.

The empty statement `skip` does nothing, i.e. it does not change state. Clearly, it is identity on state s (Fig. 6).

$$[\![\texttt{skip}]\!] = id_s \quad \text{or equivalently} \qquad [\![\texttt{skip}]\!]s = s$$

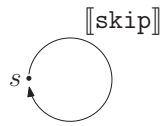

Figure 6: Morphism for empty statement

A sequence of statements is executed one by one and can be modeled as composition of morphisms (Fig 7)

$$[\![S_1; S_2]\!] = [\![S_2]\!] \circ [\![S_1]\!]$$

and defined for a state s by

$$[\![S_1; S_2]\!]s = [\![S_2]\!]\,([\![S_1]\!])\,s.$$

Because the body of a program is a sequence of statements, program semantics is a path in category $\mathscr{C}_{State}$. If the state s is undefined, i.e. $s = s_\perp$, then execution of any statement in s provides also undefined state:

$$[\![S]\!]s_\perp = s_\perp.$$

From this definition follows that achieving undefined state $s_\perp$ is similar as falling into "black hole". It means that execution of program is immediately stopped without resulting state.

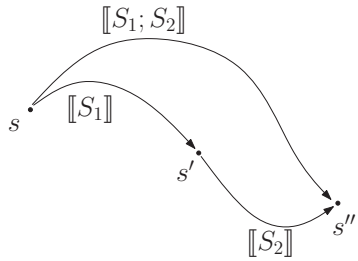

Figure 7: Composition of morphisms

Conditional statement

$$\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2$$

causes branching of execution depending on a value of Boolean expression:

$$[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]s = \begin{cases} [\![S_1]\!]s, & \text{if } [\![b]\!]s = \mathbf{true} \\ [\![S_2]\!]s, & \text{otherwise} \end{cases}$$

The path of a program can follow either to the state $[\![S_1]\!]s$ or to the state $[\![S_2]\!]s$, resp., deterministically as it is illustrated in Figure 8.

Figure 8: Execution of conditional statement

A cycle

$$\texttt{while } b \texttt{ do } S$$

also depends on a value of Boolean expression b. If b is true in initial state, i.e. $[\![b]\!]s = \mathbf{true}$, the body S of a cycle is executed, then again b is evaluated in modified state. If a value b is not valid, $[\![b]\!]s = \mathbf{false}$, then execution of a cycle statement is finished without execution of the body S. Cycle statement is semantically equivalent with the following conditional statement:

$$[\![\texttt{while } b \texttt{ do } S]\!]s = \\ [\![\texttt{if } b \texttt{ then } (S; \texttt{while } b \texttt{ do } S) \texttt{ else skip}]\!]s$$

The proof can be found in [21]. Therefore, the semantics of the cycle statement is represented in $\mathscr{C}_{State}$ as a (possibly infinite) sequence of morphisms.
When modeling execution of this statement in the category $\mathscr{C}_{State}$, two situations can appear. The first situation depicted in Figure 9 comes when the execution of cycle statement finishes in some state s_n.

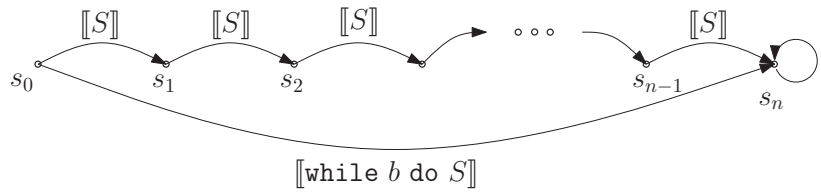

Figure 9: Terminated execution of cycle statement

However, if the condition b has always value **true**, the cycle statement is executed as infinite path in category $\mathscr{C}_{State}$. In this case, no result state is provided by the cycle statement.

Input statement `input` x serves for reading input value that is stored in the given variable x. Because the value of variable is changed, execution of input statement causes modification of state. If a variable x is not declared, the final state is undefined:

$$[\![\texttt{input } x]\!]s = \begin{cases} s\left[((x,l),v) \mapsto ((x,l),v')\right], & \text{for } ((x,l),v') \in s; \\ \bot, & \text{otherwise.} \end{cases}$$

Programs in $\mathscr{J}ane$ can have nested blocks together with declarations of local variables. Execution of block statement

$$\texttt{begin } D; S \texttt{ end}$$

follows in the following steps:

- nesting level l is incremented. We represent this step by fictive new entry in state table
$$((\texttt{begin}, l+1), \bot)$$
i.e. endomorphism $s \to s$;
- local declarations are elaborated on new nesting level $l+1$;
- the body S of block is executed;
- locally declared variables are forgotten at the end of block. We model this situation using operation $[\![del]\!]$.

From the previous points, the semantics of block statement is the following composition of morphisms:

$$[\![\texttt{begin } D; S \texttt{ end}]\!]s = \\ [\![del]\!] \circ [\![S]\!] \circ [\![D]\!](s \diamond \langle((\texttt{begin}, l+1), \bot)\rangle)$$

It follows from the construction of category of states fulfilling of its base properties:

- each object has identity morphism defined;
- for any two composable morphisms there exists such a morphism which is their composition.

We can state that $\mathscr{C}_{State}$ constructed above is a category.

8. Modeling a simple program execution in $\mathscr{C}_{State}$

We show our approach on a simple example. We consider here a trivial program written in $\mathscr{J}ane$. An absolute value of subtracting two numbers stored in variables is calculated. The program tests whether the first number is greater than the second one. If not, then local block is created with one auxiliary declared variable and the values in variables are exchanged. After that, the subtraction is executed.

```
var x; var y; var d;
input x;
input y;
if (x ≤ y) then
      begin
        var z;
        z := x;
        x := y;
        y := z;
      end;
else
   skip;
d := x − y;
```

Following the definitions in the sections 6 and 7, states changes during execution of this program are in Figure 10. Figure 11 shows the path how the given program is executed step-by-step from the initial state s_0 to the final state s_7.

Program starts from the initial state with the declaration of three variables: x, y and d. This is expressed by the composition of the endomorphisms over state s_0 (Fig. 10, part a)). After initial declarations two statements for user input are being executed. Here we assume that user inputs value **2** into variable x (Fig. 10, part b)) and value **7** into variable y (Fig. 10, part c)).

The next step is the evaluation of Boolean condition in `if`-statement. Because the condition is true, $[\![x \leq y]\!]s_2 = \mathbf{true}$, program continues by creating a new block: the fictive entry with the new level of declaration is written into state table. After that, a new variable z is being declared, represented by new endomorphism over the state s_2. These two steps appear still in state s_2 and no new state is created.

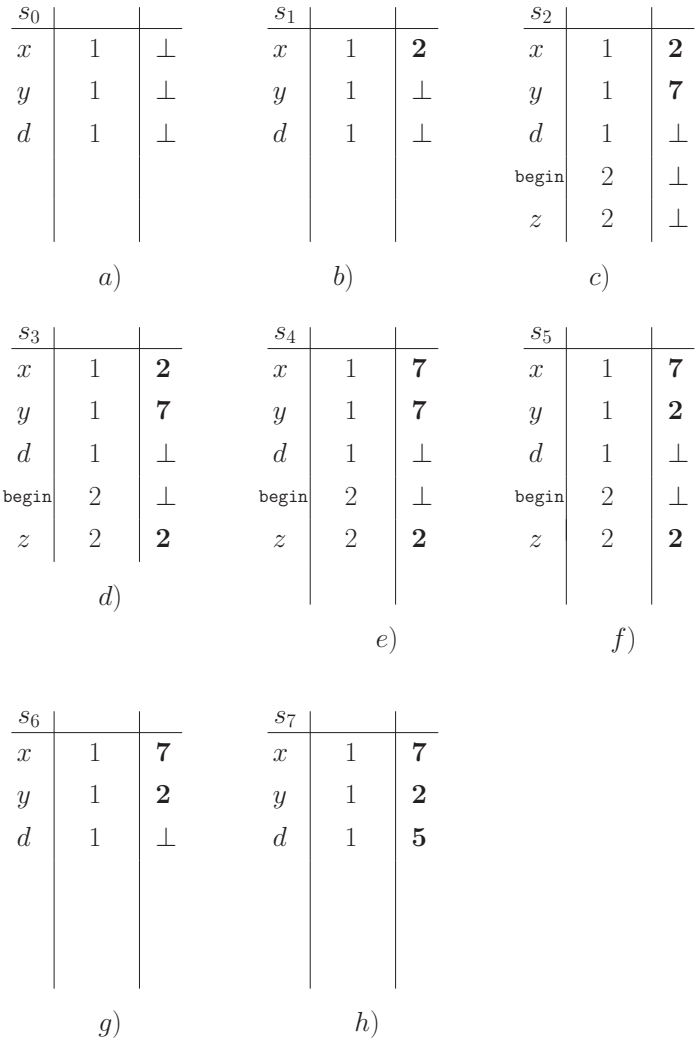

Figure 10: State changes during the program execution

The next three steps are morphisms which represent three assignment statements. New states become: state s_3 after the first assignment (Fig. 10, part d)), state s_4 after the second assignment (Fig. 10, part e)) and state s_5 after the third assignment (Fig. 10, part f)) inside the block.

When the next step is finishing of the local block, an operation for deleting all locally declared variables is being call represented by morphism $[\![del]\!]$ and the resulting state is s_6 (Fig. 10, part g)). We notify that semantics of the whole conditional statement is a composition of morphisms starting in state s_2 and finishing in state s_6 when the condition of conditional statement is true (in this case).

Finally, the last statement of assignment is executed and output of this morphism is also

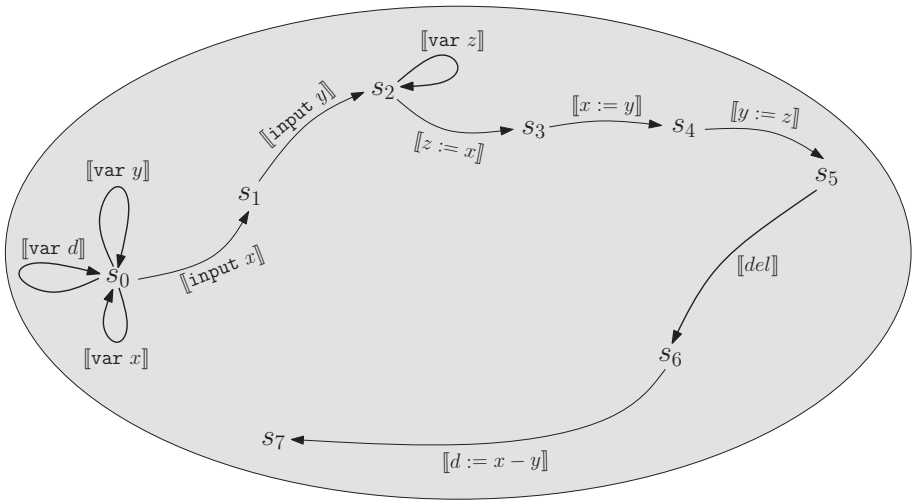

Figure 11: The path of execution of given program

the resulting state of the whole program - state s_7 (Fig. 10, part h)).

9. Conclusion

This paper contains our new approach to operational semantics by categories. We constructed the category of states $\mathscr{C}_{State}$ where states of memory are objects and state changes (computations) are morphisms. The semantics of program is defined as path that is composition of morphisms from initial state into final state. We have illustrated behavior of program on a sample example written in programming language $\mathscr{J}ane$.

Categories have beautiful illustrative power when expressing some relations graphically, our approach is very good understandable for students and also for software engineers. In the future, we would like to focus on types of data structures, exceptions, jumps and procedures. We assume that computation by procedure shall be defined in separate category. Each category that represents computation of procedure will be defined as object of total category for procedure environments.

Acknowledgments

This work has been supported by Grant No. FEI-2015-18: Coalgebraic models of component systems.

References

[1] S. Ristić, S. Aleksić, M. Čelikovic, V. Dimitrieski, and I. Luković, "Database reverse engineering based on meta-models," *Central European Journal of Com-*

puter Science, Vol. 4, No. 3, pp. 150–159, 2014.

[2] J. Labun, M. Soták, and P. Kurdel, "Technical note innovative technique of using the radar altimeter for prediction of terrain collision threats," *The Journal of the American Helicopter Society*, Vol. 57, No. 4, 2012.

[3] J. Tóth, L. Ovseník, J. Turán, and M. Tatarko, "Long term availability analysis of experimental free space optics system," in *IWSSIP 2015: 22nd International Conference on Systems, Signals and Image Processing*. City University, UK, 2015, pp. 29–32.

[4] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge, MA, USA: MIT Press, 1977.

[5] D. A. Schmidt, *Denotational semantics. Methodology for language development*. Allyn and Bacon, 1986.

[6] V. Novitzká, D. Mihályi, and V. Slodičák, "Categorical models of logical systems in the mathematical theory of programming," *Pure Mathematics and Applications*, Vol. 17, No. 3-4, pp. 367–378, 2006.

[7] ——, "Linear logical reasoning on programming," *Acta Electrotechnica et Informatica*, Vol. 6, No. 3, pp. 34–39, 2006.

[8] C. B. Jones, "Operational semantics: concepts and their expression," *Information Processing Letters*, Vol. 88, No. 1-2, pp. 27–32, 2003.

[9] G. Kahn, "Natural semantics," in *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings (LNCS 247)*. Springer-Verlag, 1987, pp. 22–39.

[10] D. A. Schmidt, "Natural-semantics-based abstract interpretation (preliminary version)," in *SAS*, Ser. Lecture Notes in Computer Science, vol. 983. Springer, 1995, pp. 1–18.

[11] R. Bagnara, P. M. Hill, A. Pescetti, and E. Zaffanella, "Verification of C programs via natural semantics and abstract interpretation," in *Proceedings of the C/C++ Verification Workshop*, Oxford, UK, 2007, pp. 75–80.

[12] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, Vol. 12, No. 10, pp. 576–580, 1969.

[13] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1, 2*, Ser. EATCS Monographs on Theoretical Computer Science. Berlin, Heidelberg, New York, Tokio: Springer-Verlag, New York, 1985, 1990.

[14] M. Wirsing, "Handbook of theoretical computer science (vol. b)." Cambridge, MA, USA: MIT Press, 1990, ch. Algebraic Specification, pp. 675–788.

[15] S. Abramsky, "Semantics of interaction: an introduction to game semantics," in *Proceedings of the 1996 CLiCS Summer School, Isaac Newton Institute*. Cambridge University Press, 1997, pp. 1–31.

[16] P. Curien, "Notes on game semantics," 2006, http://www.pps.jussieu.fr/ curien/Game-semantics.pdf - Electronic Edition.

[17] G. D. Plotkin, "A structural approach to operational semantics," University of Aarhus, Tech. Rep. DAIMI FN-19, 1981.

[18] ——, "The origins of structural operational semantics," *The Journal of Logic and Algebraic Programming*, Vol. 60-61, pp. 3–15, 2004.

[19] D. Turi and G. Plotkin, "Towards a mathematical operational semantics," in *In Proc. 12 th LICS Conf.* IEEE, Computer Society Press, 1997, pp. 280–291.

[20] L. Aceto, W. J. Fokkink, and C. Verhoef, "Structural operational semantics," in *Handbook of Process Algebra*. Elsevier, 1999, pp. 197–292.

[21] F. Nielson and H. R. Nielson, *Semantics with applications. A formal introduction*. Wiley and Sons, 1992.

[22] D. Turi, "Functorial operational semantics and its denotational dual (Ph.D. thesis)," Ph.D. dissertation, University of Amsterdam, 1996.

[23] D. A. Schmidt, "Abstract interpretation of small-step semantics," in *Proceedings of the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages. LNCS 1192*. Springer-Verlag, 1996, pp. 76–99.

[24] ——, "Trace-based abstract interpretation of operational semantics," *LISP and Symbolic Computation*, Vol. 10, No. 3, pp. 237–271, 1998.

[25] M. Jaskelioff, N. Ghani, and G. Hutton, "Modularity and implementation of mathematical operational semantics," *Electronic Notes in Theoretical Computer Science*, Vol. 229, No. 5, pp. 75–95, 2011, Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008).

[26] P. Mosses, "Modular structural operational semantics," *Journal of Logic and Algebraic Programming*, Vol. 60-61, 2004.

[27] S. Staton, "General structural operational semantics through categorical logic," in *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science (LICS 2008)*. IEEE Computer Society Press, 2008, pp. 166–177.